参考链接： 编程风格 - ECMAScript 6入门 (ruanyifeng.com)
原文链接：https://blog.csdn.net/Noria107/article/details/113057241

1、Class

Class：类，对象的模板；ES6 的 class 可以看作只是 一个语法糖，它的绝大部分功能，ES5 都可以做到，新的 class 写法只是让对象原型的写法更加清晰、更像面向对象编程的语法。
语法：

- class：声明类
- constructor：定义构造函数初始化

```
<script>
        // 1. 创建类 class  创建一个 明星类
        class Star {
            constructor(uname, age) {
                this.uname = uname;
                this.age = age;
            }
        }

        // 2. 利用类创建对象 new
        var ldh = new Star('刘德华', 18);
        var zxy = new Star('张学友', 20);
        console.log(ldh);
        console.log(zxy);
        //(1) 通过class 关键字创建类, 类名我们还是习惯性定义首字母大写
        //(2) 类里面有个constructor 函数,可以接受传递过来的参数,同时返回实例对象
        //(3) constructor 函数 只要 new 生成实例时,就会自动调用这个函数, 如果我们不写这个函数,类也会自动生成这个函数
        //(4) 生成实例 new 不能省略
        //(5) 最后注意语法规范, 创建类 类名后面不要加小括号,生成实例 类名后面加小括号, 构造函数不需要加function
    </script>
```

类中添加方法：

```
<script>
        // 1. 创建类 class  创建一个 明星类
        class Star {
            // 类的共有属性放到 constructor 里面
```

```
            constructor(uname, age) {
                this.uname = uname;
                this.age = age;
            }
            // 添加方法
            sing(song) {
                // console.log('我唱歌');
                console.log(this.uname + song);

            }
        }

        // 2. 利用类创建对象 new
        var ldh = new Star('刘德华', 18);
        var zxy = new Star('张学友', 20);
        console.log(ldh);
        console.log(zxy);
        // (1) 我们类里面所有的函数不需要写function
        // (2) 多个函数方法之间不需要添加逗号分隔
        ldh.sing('冰雨');
        zxy.sing('李香1');
    </script>
```

- extends：继承父类；extends关键字可以继承类，也可以继承原生的构造函数

```
//1 类的继承
class Father {
    constructor(){

    }
    money(){
        console.log('100');
    }
}
//extends继承
class Son extends Father{

}
let son = new Son();
```

```
son.money();
```

- super：调用父级构造方法；super作为函数调用时代表父类的构造函数

```
// super 关键字调用父类普通函数
class Father {
    say() {
        return '我是aaaa';
    }
}
class Son extends Father {
    say() {
        console.log('我是aaaa');
        console.log(super.say() + '的aaaa')
        // super.say() 就是调用父类中的普通函数 say()
    }
}
let son = new Son();
son.say();
// 继承中的属性或者方法查找原则：就近原则
// 1. 继承中,如果实例化子类输出一个方法,先看子类有没有这个方法,如果有就先执行子类的
// 2. 继承中,如果子类里面没有,就去查找父类有没有这个方法,如果有,就执行父类的这个方法(就近原则)
```

子类继承父类方法的同时拓展自己的方法：

```
//父类的加法方法
class Father {
    constructor(x,y){
        this.x = x;
        this.y = y;
    }
    sum() {
        consloe.log(this.x + this.y);
    }
}
//子类继承父类加法方法 同时 扩展减法方法
class Son extends Father{
    constructor(x,y) {
        // 利用super 调用父类的构造函数
```

```
        // super 必须在子类this之前调用
        super(x, y);
        this.x = x;
        this.y = y;
    }
    subtract() {
        console.log(this.x - this.y)
    }
}
let son = new Son(5, 3);
son.subtract();
son.sum();
```

- static：定义静态方法和属性

```
class Axios {
//静态的属性
    static defaults = {
         baseUrl: ''
    };
    constructor() {}
    static get(url, query) {
        console.log(`请求地址：${Axios.defaults.baseUrl + url}`);
        console.log(`传参：${query}`);
    }
    static post(url, body) {
        console.log(`请求地址：${Axios.defaults.baseUrl + url}`);
        console.log(`传参：${body}`);
    }
}
//通过类名直接调用静态方法
    Axios.defaults.baseUrl = 'http://jd.com'; //服务器域名
    Axios.get('/login', 'username=zhangsan&password=123456'); //Object
    Axios.post('/reg', 'username=liangzai&password=654321'); //Object
```

class 私有属性：只能在class中访问

```
class Person {
    //公有属性
```

```
    name;
    //私有属性
    #age;
    #weight;

    //构造方法
    constructor(name, age, weight) {
        this.name = name;
        this.#age = age;
        this.#weight = weight;
    }

    //普通方法
    intro() {
        console.log(this.name);
        console.log(this.#age);
        console.log(this.#weight);
    }
}

//实例化
const boy = new Person("张三", 20, "50kg");
boy.intro();
```

使用类注意事项：

```
<button>点击</button>
<script>
   let that;
   let _that;
   class Star {
       constructor(uname, age) {
           //constructor中的this指向创建的实例化对象
           that = this;
           console.log(this);
           this.uname = name;
           this.age = age;
           this.btn = document.querySelector('button');
           this.btn.onclick = this.sing;
```

```
        }
        sing() {
            //sing里面的this指向btn这个按钮，因为这个按钮调用了sing方法
            console.log(this);
            console.log(that.uname);
            //that里面存储的是constructor里面的this
        }
        dance() {
            _that = this;
            // 这个dance里面的this 指向的是实例对象 ldh 因为ldh 调用了这个函数
            console.log(this);
        }
    }
    let ldh = new Star('liudehua');
    console.log(ldh);
    ldh.dance();
    console.log(that === ldh); //true
    console.log(_that === ldh); //true
    // 1. 在 ES6 中类没有变量提升，所以必须先定义类，才能通过类实例化对象
    // 2. 类里面的共有的属性和方法一定要加this使用.
  </script>
```

类的本质

```
// ES6 之前通过 构造函数+ 原型实现面向对象 编程
// (1) 构造函数有原型对象prototype
// (2) 构造函数原型对象prototype 里面有constructor 指向构造函数本身
// (3) 构造函数可以通过原型对象添加方法
// (4) 构造函数创建的实例对象有__proto__ 原型指向 构造函数的原型对象
// ES6 通过 类 实现面向对象编程
class Star {

}
console.log(typeof Star); // function
// 1. 类的本质其实还是一个函数 我们也可以简单的认为 类就是 构造函数的另外一种写法
// (1) 类有原型对象prototype
console.log(Star.prototype);
// (2) 类原型对象prototype 里面有constructor 指向类本身
console.log(Star.prototype.constructor);
```

```
// (3)类可以通过原型对象添加方法
Star.prototype.sing = function () {
    console.log('冰雨');
}
var ldh = new Star();
console.dir(ldh);
// (4) 类创建的实例对象有__proto__ 原型指向 类的原型对象
console.log(ldh.__proto__ === Star.prototype);
```

2、Module（模块化）

模块加载方案：

- CommonJS：用于服务器
- AMD：用于浏览器

ES6模块，通过export命令显式指定输出的代码，在通过import命令输入

```
<script type="module">
        //*   代表所有的
        //as  作为：取个别名
        import * as mymodule from './module/m.js';
        console.log(mymodule);

        //解构赋值
        import {swipper, fn, a}  from './module/m.js';
        console.log(fn);
        console.log(a);
        console.log(b);
    </script>
```

模块功能主要有import和export组成

（一）export

- 规定模块的对外接口（规定的是对外的接口，必须与模块内部变量对应，否则报错）
- 可输出变量、函数、类，可使用as关键字重命名
- 可出现在模块任意位置，只要处于模块顶层即可
- 若处于块级作用域，则报错

（二）import

- 输入其他模块提供的功能
- 可出现在模块任意位置，只要处于模块顶层即可

- 若处于块级作用域，则报错
- import后面可加一个from指定模块文件的位置
- import有提升效果，会提升到整个模块的头部并首先执行
- import为静态执行，不能使用表达式和变量，只有在运行时才能得到语法结构
- 若多次重复执行用一句import语句，只会执行一次，不会多次执行
- import是Singleton模式（调用一个类，并在任何时候都返回一个实例）
- 可使用export default为模块指定默认输出（只能用一次）
- 模块也可以继承

(三)加载

浏览器在加载ES6模块是，使用

```
<script type="module" src="... .js"></script>
```

模块化的暴露:

```
//方式一：分别暴露
export let school = "aaa";

export function study() {
    console.log("AAA");
}
// 接收
<script type="module">
// 方式一
import school from './js/...';
import * as study from './js/...';

// 方式二
import {school,study} from './js/...';
</script>

```

```
//方式二：统一暴露
let school = "bbb";

function findJob() {
    console.log("BBB");
```

```
}

export {school, findJob};
```

```
//方式三：默认暴露
export default {
    school: "ccc",
    change: function () {
        console.log("CCC");
    }
}
```

import导入：

```
在html中导入：
<script type="module">
    import Swipper1 from './js/swipper.class.js';
    //console.log(moverFn);
    let p = new Swipper1('#mylunbo');
    p.autoplay();
    console.log(p);
 </script>
```

动态 import导入：

```
<body>

<button id="btn">点击我，加载m1.js模块</button>

<!-- 在这里写JavaScript代码，因为JavaScript是由上到下执行的 -->
<script type="module">
    const btn = document.getElementById("btn");

    btn.onclick = function(){
        import("./m1.js").then(module => {
            module.study();
        });
    };
```

```
</script>
```

3、编程风格

let完全取代var（let无副作用）

const优于let的原因：

- const提醒这个变量不可变
- const符合函数式编程思想，运算不改变值，只是新建值，有利于分布式运算
- JS编译器会对const优化处理，有利于提高程序的运行效率

let和const的本质区别是编译器内部处理不同

静态字符串一律使用单引号或者反引号，不使用双引号，动态字符串使用反引号

使用数组成员对变量赋值时，优先使用解构赋值

若函数的参数为对象的成员，优先使用解构赋值

若函数有多个返回值，优先使用解构赋值

单行定义对象，最后一个成员不以逗号结尾

多行定义对象，最后一个成员要以逗号结尾

若对象的属性名为动态的，可在创造对象时使用属性表达式定义

使用扩展运算符（...）复制数组

立即执行的函数写为箭头函数形式

在函数体内使用rest运算符（...）代替arguments变量

使用Class代取prototype操作

用extends继承

使用import代替require：

```
import {function1,function2} from 'Module';
```

使用export 代替module.exports：

```
import React from 'react';
class Bread extends React.Compont{
render(){
return ;
   }
}
export default Bread;
```

若模块只有一个输出值，则需使用export default

不能模块中使用通配符

若模块默认输出一个函数，则函数名首字母需小写
若模块默认输出一个对象，则对象名首字母需大写

4、模块化编程案例

html代码：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>轮播图</title>
    <link rel="stylesheet" href="./css/index.min.css">
</head>
<body>
    <div class="swipper" id="mylunbo">
        <!-- 视觉视口 -->
        <div class="view">
            <!-- 布局视口 -->
            <ul class="lay">
                <li>
                    <img src="https://www.lpssy.edu.cn/_upload/article/images/fd/c4/f0e632ab4067973d74606f8a1958/3e50d1d6-fafa-446c-a038-0fbb5cb91936.jpg" alt="">
                </li>
                <li>
                    <img src="https://www.lpssy.edu.cn/_upload/article/images/93/03/84cc6e26441bb556961fe37199df/7d9983a0-774f-4b3a-b9c6-c328a0abb749.jpg" alt="">
                </li>
                <li>
                    <img src="https://www.lpssy.edu.cn/_upload/article/images/fd/c4/f0e632ab4067973d74606f8a1958/3e50d1d6-fafa-446c-a038-0fbb5cb91936.jpg" alt="">
                </li>
                <li>
                    <img src="https://www.lpssy.edu.cn/_upload/article/images/93/03/84cc6e26441bb556961fe37199df/7d9983a0-774f-4b3a-b9c6-c328a0abb749.jpg" alt="">
                </li>
```

```
                <li>
                    <img
src="https://www.lpssy.edu.cn/_upload/article/images/fd/c4/f0e632ab4067973d74606f8a1958/3e50d1d6-fafa-446c-a038-0fbb5cb91936.jpg" alt="">
                </li>
            </ul>
        </div>
        <ul class="point">
           <li></li>
           <li></li>
           <li></li>
           <li></li>
           <li></li>
        </ul>
    </div>

    <script type="module">
        import Swipper1 from './js/swipper.class.js';
        //console.log(moverFn);
        let p = new Swipper1('#mylunbo');
        p.autoplay();
        console.log(p);
    </script>
</body>
</html>
```

sass代码：

_variable.scss

```
$swipper-width: 1600px;
$swipper-height: 480px;
```

_common.scss

```
*{
    margin: 0;
    padding: 0;
}
body{
```

```
    font: 14px/1 Microsoft YaHei, Helvetica, STHeiti STXihei, Microsoft JhengHei, Tohoma, Arial;
    color: #333;
}
li{
    list-style: none;
}

```

_swipper.scss

```
.swipper {
    width: $swipper-width;
    background-color: #f01;
    margin: 100px auto;
    height: $swipper-height;
    // 子绝父相
    position: relative;
    > .view {
        width: 100%;
        height: 100%;
        background-color: #00f;
        overflow: hidden;
        > .lay {
            width: 500%;
            height: 100%;
            background-color: #0f0;
            display: flex;
            >li{
                width: $swipper-width;
            }
        }
    }
    > .point {
        position: absolute;
        width: 100%;
        height: 42px;
        bottom: 20px;
        display: flex;
        justify-content: center;
```

```
        align-items: center;
        > li {
            width: 18px;
            height: 18px;
            border-radius: 50%;
            -webkit-border-radius: 50%;
            -moz-border-radius: 50%;
            -ms-border-radius: 50%;
            -o-border-radius: 50%;
            background-color: #fff;
            margin-left:15px ;
            cursor: pointer;
        }
    }
}

```

index.scss

```
@import './module/variable';
@import './module/common';
@import './module/swipper';
```

CSS代码：

```
* {
  margin: 0;
  padding: 0;
}

body {
  font: 14px/1 Microsoft YaHei, Helvetica, STHeiti STXihei, Microsoft JhengHei, Tohoma, Arial;
  color: #333;
}

li {
  list-style: none;
}
```

```
14
15 .swipper {
16   width: 1600px;
17   background-color: #f01;
18   margin: 100px auto;
19   height: 480px;
20   position: relative;
21 }
22
23 .swipper > .view {
24   width: 100%;
25   height: 100%;
26   background-color: #00f;
27   overflow: hidden;
28 }
29
30 .swipper > .view > .lay {
31   width: 500%;
32   height: 100%;
33   background-color: #0f0;
34   display: flex;
35 }
36
37 .swipper > .view > .lay > li {
38   width: 1600px;
39 }
40
41 .swipper > .point {
42   position: absolute;
43   width: 100%;
44   height: 42px;
45   bottom: 20px;
46   display: flex;
47   justify-content: center;
48   align-items: center;
49 }
50
51 .swipper > .point > li {
52   width: 18px;
53   height: 18px;
```

```
  border-radius: 50%;
  -webkit-border-radius: 50%;
  -moz-border-radius: 50%;
  -ms-border-radius: 50%;
  -o-border-radius: 50%;
  background-color: #fff;
  margin-left: 15px;
  cursor: pointer;
}
/*# sourceMappingURL=index.min.css.map */
```

js代码：

swipper.function.js

```
// 构造函数一般是进行属性的初始化
function Swipper(selector) {
    //拿到所有的事件源--》圆点
    this.allPoints = [...document.querySelectorAll(`${selector}>.point>li`)];
    //要操作的布局视口
    this.lay = document.querySelector(`${selector}>.view>.lay`);
    //setup
    this.setup = 1600;
    //初始位置
    this.mleft = 0;
    //当前视觉视口显示的图片下标
    this.ind = 0;
    //轮播图初始化
    this.init();
}
Swipper.prototype.init = function () {
    const $this = this;
    //通过循环的当时给每个点绑定点击事件
    this.allPoints.forEach(function (p, i) {
        p.onclick = function () {
            $this.imgmove(i);
        }
    });
}
Swipper.prototype.imgmove = function (i) {
```

```
    const $this = this;
    // 改变布局视口的margin-left
    let to = -this.setup * i;
    let sid = setInterval(function () {
        //判断点击的点在前面还是后面
        if (i > $this.ind) {
            $this.mleft -= 25;
        } else {
            $this.mleft += 25;
        }
        console.log($this.mleft);
        $this.lay.style.marginLeft = $this.mleft + 'px';
        if ($this.mleft === to) {
            //清除定时器
            clearInterval(sid);
            //重置当前视觉视口里面的显示的图片下标
            $this.ind = i;
        }
    }, 5);
}
export default Swipper;
```

swipper.class.js

```
const moverFn = Symbol();//sync45678dasdasf45weretwerteer
//使用symbol实现私有的属性或方法
export default class Swipper {
    // 构造函数一般是进行属性的初始化
    constructor(selector) {
        //拿到所有的事件源--》圆点
        this.allPoints = [...document.querySelectorAll(`${selector}>.point>li`)];
        //要操作的布局视口
        this.lay = document.querySelector(`${selector}>.view>.lay`);
        //setup
        this.setup = 1600;
        //初始位置
        this.mleft = 0;
        //当前视觉视口显示的图片下标
        this.ind = 0;
```

```
        //定时器id
        this.sid = 0;
        //布局视口的左边距
        this.to = 0;
        //轮播图初始化
        this.init();
    }
    // 轮播图的初始化-->点击事件绑定
    init() {
        const $this = this;
        //通过循环的当时给每个点绑定点击事件
        this.allPoints.forEach(function (p, i) {
            p.onclick = function () {
                $this[moverFn](i);
            }
        });
    }
    //让图动起来  -->sync45678dasdasf45weretwerteer
    [moverFn](i) {
        const $this = this;
        //定时器冲突问题
        if ($this.sid) {
            clearInterval($this.sid);
            $this.lay.style.marginLeft = $this.to + 'px';
            $this.mleft = $this.to;
        }
        // 改变布局视口的margin-left
        $this.to = -this.setup * i;
        $this.sid = setInterval(function () {
            //判断点击的点在前面还是后面
            if (i > $this.ind) {
                $this.mleft -= 25;
            } else {
                $this.mleft += 25;
            }
            $this.lay.style.marginLeft = $this.mleft + 'px';
            if ($this.mleft === $this.to) {
                //清除定时器
                clearInterval($this.sid);
                //重置当前视觉视口里面的显示的图片下标
```

```
                $this.ind = i;
            }
        }, 5);
    }
    //自动轮播
    autoplay() {
        const $this = this;
        setInterval(function () {
            $this.allPoints[($this.ind + 1) % $this.allPoints.length].click();
        }, 3000);
    }
}
```